Maths3D

Maths for Computer Graphics

John Ryland

Sunday 15th May, 2022

Contents

List of Figures

1 Maths3D

1.0.1 Maths for Computer Graphics

1.1 Status

Coverage Report

Documentation

PDF

Source

Releases

1.2 Introduction

This code requires a modern C++ compiler but is mostly C and avoids using classes (with the exception of providing unit type safety discussed below).

No header or library dependencies.

The API attempts to be functional in style, with inputs and a returned output.

The inputs are never mutated (modified) and the returned value is a newly constructed object. This is to maintain referential transparency. Some benefits to this approach are that it is easy to chain together calls to these APIs. It also means that they can be safely used in multithreaded code. From a performance point of view it assumes that the compiler will be able to perform RVO.

Inputs are passed by const reference when the size of the input would be larger than the size of a pointer.

2 Scalars

Changing the scalar type will change the underlying value type used by vector and matrix.

A scalar value used to represent an angle can sometimes be used incorrectly when an API is not clear if the units are degrees or radians. To avoid this, the API itself should document the units with a specific type that encode the units. The types Degrees and Radians below do this. These are actually classes and so this is C++, but it is worth the developer time it saves by ensuring there is no mistakes of this kind. The same could be used for other types of units as required, such as meters vs feet if the API were to need this.

3 Vectors

3.1 Vector Components

Both the vector and matrix type contain unions with a 'v' member which can be used for iteration of the components instead of needing to write code which requires copy paste of the same code for each component. This avoids mistakes, such as:

```
Vector4f ret;
ret.x = a.x + b.x;
ret.y = a.y + b.y;
ret.y = a.z + b.z;
```

Do you see the mistake? Happens all the time. Far better is to write code like this:

```
Vector4f ret;
for (int i = 0; i < 3; ++i)
  ret.v[i] = a.v[i] + b.v[i];
```

Less lines of code, less error prone, and easier to update the formula applied to each component.

3.2 Vector width

The API omits providing a 3 component wide vector and instead only provides a Vector4f. This is because when dealing with arrays of these, we will be able to optimize better the 4 wide version, particularly if it can be aligned to 128-bits and used with a SIMD optimized function to transform an array of them. For all the provided functions below, we don't bother providing SIMD optimizations as most compilers can auto-vectorize this code just fine and so it only will make the code more error prone, and not able to be re-compiled with different compiler flags to target different machine types or without or without different levels of SIMD support.

4 The API

```
// Vector functions
Vector4f Vector4f_Set(Scalar1f x, Scalar1f y, Scalar1f z, Scalar1f w);
Vector4f Vector4f_Replicate(Scalar1f v);
Vector4f Vector4f_Zero();
Vector4f Vector4f_SetX(const Vector4f& vec, Scalar1f x);
Vector4f Vector4f_SetY(const Vector4f& vec, Scalar1f y);
Vector4f Vector4f_SetZ(const Vector4f& vec, Scalar1f z);
Vector4f Vector4f_SetW(const Vector4f& vec, Scalar1f w);

// Vector operations
Vector4f Vector4f_CrossProduct(const Vector4f& v1, const Vector4f& v2);
Vector4f Vector4f_Multiply(const Vector4f& vec1, const Vector4f& vec2);
Vector4f Vector4f_Add(const Vector4f& vec1, const Vector4f& vec2);
Vector4f Vector4f_Scaled(const Vector4f& vec, Scalar1f scale);
Scalar1f Vector4f_SumComponents(const Vector4f& vec);
Scalar1f Vector4f_DotProduct(const Vector4f& vec1, const Vector4f& vec2);
Scalar1f Vector4f_LengthSquared(const Vector4f& vec);
Scalar1f Vector4f_Length(const Vector4f& vec);
Scalar1f Vector4f_ReciprocalLength(const Vector4f& vec);
Vector4f Vector4f_Normalized(const Vector4f& vec);

// Matrix functions
Matrix4x4f Matrix4x4f_Set(const Scalar1f v[16]);
Matrix4x4f Matrix4x4f_Zero();
Matrix4x4f Matrix4x4f_Identity();

// Matrix transforms
Matrix4x4f Matrix4x4f_TranslateXYZ(const Vector4f& vec);
Matrix4x4f Matrix4x4f_Multiply(const Matrix4x4f& m1, const Matrix4x4f& m2);
Matrix4x4f Matrix4x4f_Transposed(const Matrix4x4f& a);
Matrix4x4f Matrix4x4f_Scaled(const Matrix4x4f& m, Scalar1f scale);
Matrix4x4f Matrix4x4f_ScaleXYZ(const Vector4f& scale);
Matrix4x4f Matrix4x4f_RotateX(Scalar1f x);
Matrix4x4f Matrix4x4f_RotateY(Scalar1f y);
Matrix4x4f Matrix4x4f_RotateZ(Scalar1f z);
Vector4f Vector4f_Transform(const Matrix4x4f& m, const Vector4f& vec);

// Projections
Matrix4x4f Matrix4x4f_PerspectiveFrustum(Radians fieldOfView, Scalar1f aspectRatio,
                                         Scalar1f near, Scalar1f far);
Matrix4x4f Matrix4x4f_OrthographicFrustum(Scalar1f left, Scalar1f right,
                                          Scalar1f bottom, Scalar1f top,
                                          Scalar1f near, Scalar1f far);
```